Efficient Test Generation Guided by Field Coverage Criteria

Ariel Godio
Department of Software Engineering
ITBA
Buenos Aires, Argentina
agodio@itba.edu.ar

Valeria Bengolea
Department of Computer Science
UNRC
Rio Cuarto, Argentina
vbengolea@dc.exa.unrc.edu.ar

Pablo Ponzio
Department of Computer Science
UNRC
Rio Cuarto, Argentina
pponzio@dc.exa.unrc.edu.ar

Nazareno Aguirre
Department of Computer Science
UNRC
Rio Cuarto, Argentina
naguirre@dc.exa.unrc.edu.ar

Marcelo F. Frias
Department of Software Engineering
ITBA
Buenos Aires, Argentina
mfrias@itba.edu.ar

***Abstract*—Field-exhaustive testing is a testing criterion suitable for object-oriented code over complex, heap-allocated, data structures. It requires test suites to contain enough test inputs to cover *all* feasible values for the object's fields within a certain scope (input-size bound). While previous work shows that field-exhaustive suites can be automatically generated, the generation technique required a formal specification of the inputs that can be subject to SAT-based analysis. Moreover, the restriction of producing *all* feasible values for inputs' fields makes test generation costly.**

In this paper, we deal with *field coverage* as testing criteria that measure the quality of a test suite in terms of coverage and mutation score, by examining to what extent the values of inputs' fields are covered. In particular, we consider field coverage in combination with test generation based on symbolic execution to produce underapproximations of field-exhaustive suites, using the Symbolic Pathfinder tool. To underapproximate these suites we use *tranScoping*, a technique that estimates characteristics of yet to be run analyses for large scopes, based on data obtained from analyses performed in small scopes. This provides us with a suitable condition to prematurely stop the symbolic execution.

As we show, tranScoping different metrics regarding field coverage allows us to produce significantly smaller suites using a fraction of the generation time. All this while retaining the effectiveness of field exhaustive suites in terms of test suite quality.

***Index Terms*—Field-exhaustive testing, field-based testing, symbolic execution, transcoping**

I. Introduction

The functional correctness of software systems, i.e., the degree to which a software system meets the purpose for which it has been built, is among the most challenging problems in software engineering [12], [14]. While many techniques and methodologies have been proposed to guarantee software correctness, *testing* (consisting in evaluating the software under analysis by executing it in a number of cases) is by far the most widely used, product-centric, technique for software correctness assurance [2]. An essential part of testing is selecting the cases in which the software under test is going to be run. This process is very costly in practice and is performed mostly manually [2]. At the same time, its importance is acknowledged by software development methodologies that strongly encourage writing tests, e.g., agile methods supporting writing tests together with (and even before) the software that should be tested [4].

It is well known that tests are inherently incomplete as a measure for guaranteeing functional correctness. Thus, testing software *appropriately*, i.e., attempting to cover as much as possible of software functionality with a finite (and limited) set of test cases, is crucial. Different *coverage criteria*, i.e., constraints on how a set of tests should exercise software, have been proposed [30], and some have been adopted by development models and software certifications (e.g., [20]). These constraints are in many cases very difficult to comply with, making manual test development even more complex and time-consuming, especially in certain application domains. This situation has led to the active development of *automated test generation techniques*, a family of approaches that attempt to produce tests automatically. These techniques vary in their underlying automation approaches, including random generation [6], [18], [21], generation based on constraint-solving [1], [27], and generation based on search [5] (including evolutionary approaches [9]). These techniques can also be categorized as *white box* or *black box*. White box techniques are generally driven by code coverage, with tools based, e.g., in symbolic, concrete, or "concolic" execution [27], [28]; these tend to produce relatively small suites, with every test covering exactly one path, branch, or whatever the "unit" for coverage is driven by. Black box automated test generation techniques, on the other hand, generally produce *large* suites, as witnessed by tools based on random generation [6], [18], [21] and constraint solving or (exhaustive) search [5], [16]. Finally, evolutionary computation [9], whose categorization as white box or black box depends on the fitness function used, also produces large suites. This latter observation has some negative implications. For some tools, the size of the

produced suites affects the efficiency of the technique (e.g., degradation in test generation using random testing, or high cost of suite minimization in evolutionary computation based generation). And, more importantly, large test suites imply a high cost of test execution, and unsuitability in contexts where suites are continuously and repeatedly executed (e.g., in test-driven development, or continuous integration environments).

The above-mentioned problems of black-box driven test input generation motivated the recent work on a testing criterion called *Field-exhaustive testing* [23]. This black-box criterion is particularly well-suited for object-oriented code over complex, heap-allocated, data structures, and requires suites to contain enough test inputs to cover *all* feasible values for object fields within a certain input-size bound. While the criterion comes equipped with a technique for automatic generation of field-exhaustive suites, this technique requires a formal specification of the inputs that can be subject to SAT-based analysis [23]. Moreover, the restriction of producing *all* feasible values for object fields, makes test generation costly and field-exhaustive testing difficult to generalize to further testing domains.

In this paper, we deal with *field coverage* as testing criteria that measure test suite quality by examining to what extent the values of inputs' fields are covered. Intuitively, the approach we propose receives as input a family of sets of pairs

$$\mathit{field}1 = \{\langle \mathit{obj1}, \mathit{fieldVal}1\rangle, \langle \mathit{obj2}, \mathit{fieldVal}2\rangle, \ldots\},$$
$$\mathit{field}2 = \{\langle \mathit{obj1}, \mathit{fieldVal}3\rangle, \langle \mathit{obj2}, \mathit{fieldVal}4\rangle, \ldots\},$$
$$\vdots$$

Each set (for instance set $\mathit{field}1$), describes a set of pairs to cover. For example, if pair $\langle o, a\rangle \in \mathit{field}1$, then a suite satisfying field coverage must contain a data structure in which object o must be a part, and for which $o.\mathit{field}1 = a$.

In order to determine the pairs covered by a test suite, we will look at inputs (heap-stored data structures), as *rooted, labeled, graphs*. The graph root is the receiver object, and arcs relating objects are labeled with field names. In this setting, an input is considered redundant if the relations among objects that it establishes have all been covered by previous inputs. Figure 1 presents an example using three different binary trees. Nodes are indexed `T0`, `N0`, `N1` and `N2`. Above the trees, we include the relations between nodes that each tree establishes, partitioned according to field names. The grayed out arcs and pairs between objects correspond to those covered by previous inputs (considering a left to right generation order).

We show that this notion generalizes field-exhaustive testing, withdrawing the need for a SAT-analyzable formal specification, and thus can be combined with any test generation technique to produce smaller test suites. In particular, we consider field coverage in combination with test input generation based on symbolic execution of repOk() procedures that code data structure representation invariants. Since symbolic execution may be expensive, we will produce underapproximations of adequate test suites using the Symbolic Pathfinder tool. To underapproximate these suites we use tranScoping

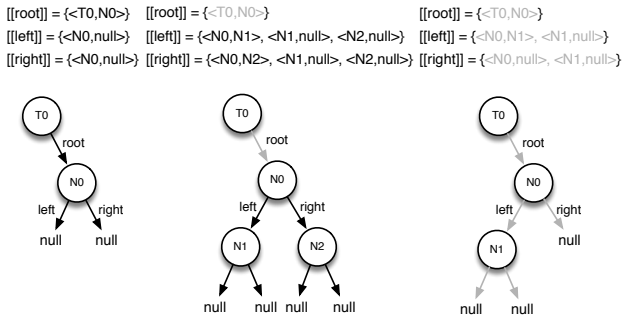

Fig. 1. Viewing heap-stored data structures as rooted, labeled, graphs.

[24], a technique that estimates characteristics of yet to be run analyses for large scopes, based on data obtained from analyses performed in small scopes. This imposes a suitable condition to prematurely stop symbolic execution. As we show, tranScoping different metrics regarding field coverage helps us produce significantly smaller suites using a fraction of the generation time. All this while retaining the effectiveness of field exhaustive suites, in terms of test suite quality.

We show that this criterion, when combined with test generation based on symbolic execution and tranScoping, improves testing as follows:

- It produces test suites that are well suited for testing programs with complex, heap-allocated, inputs.
- Significantly improves the time of the corresponding test generation technique, when compared to the generation of field-exhaustive test suites. The experiments we will report in Section IV show, for the TreeSet subject, a speedup of 167X.
- It produces test suites whose bug-detection ability is comparable to that of field-exhaustive suites while improving testing efficiency with its significantly fewer tests. Moreover, if we approximate bug-detection by measuring branch coverage and mutation score, for the TreeSet subject in Section IV we achieve the same coverage and mutation score is slightly smaller (72.4% instead of 72.6%).

The paper is organized as follows. In Section II we introduce field-coverage criteria as a class of coverage criteria and focus on field-exhaustive suites [23] and the weaker versions to be discussed in this paper. In Section III we show that suites satisfying field-coverage can be automatically generated using symbolic execution. In Section IV we present experimental results that support the applicability of the presented criteria for bug-finding. Finally, in Section VI we present some concluding remarks on this work.

II. Field-Coverage Testing Criteria

Testing criteria define testing standards that indicate how thoroughly a piece of code is being examined by test cases. Moreover, when testing criteria are selected as quality targets, they serve as a measure for testing engineers on how to proceed to test a given piece of code. Engineers typically

design testing criteria that, when met, have as a consequence good testing properties. For instance, a suite that fails to execute a certain program statement cannot detect failures originating in that statement; *statement coverage* is a testing criterion that attempts to avoid such omissions by prescribing that one should include test cases that exercise *all* the executable statements in the program under test. Notice also that certain criteria help to build test suites able to detect certain classes of errors. For instance, multiple condition coverage takes into consideration all possible combinations of outcomes in conditions within program decision points, thus aiming at logical errors or oversights in those points, affecting the desired program control flow.

Field coverage criteria aim at detecting errors in code operating on data structures, as for instance those arising in code that handles heap-allocated objects with a complex structure, or that operate on heavily constrained memory heaps. In this section, we will provide a definition of the memory heaps that programs under test are going to handle, and afterward introduce field coverage criteria.

Let

$$\mathsf{C}'\ m(arg_1 : \mathsf{C}_1, \ldots, arg_k : \mathsf{C}_k)$$

be the method under test, declared in class C, where $arg_1, \ldots, arg_k$ are the formal parameters for m, $\mathsf{C}_1, \ldots, \mathsf{C}_k$ their respective types and C' the return type of m.

Definition 1: An *input memory heap for method* m (or simply, *memory heap*, when the method under analysis is clear from the context), is a labeled graph $\langle V, E, L, r, p_1, \ldots, p_k, s_1, \ldots, s_n \rangle$ where:

- V is a set of objects and values, containing the value $null$. Objects are typed, and a single object may have more than one type.
- E is a set of labeled edges between members of V. L, the labeling function, assigns class field names to edges. $L(o_1 \rightarrow o_2) = f \iff o_1.f = o_2$. We assume the labelling function respects the heap typing.
- $r \in V$ is the receiver object, from class C.
- $p_1, \ldots, p_k \in V$ are the actual parameters for m.
- $s_1, \ldots, s_n \in V$ are the static objects/values declared in class C.

Before defining field coverage criteria, we need to introduce some notation about the semantics we will use. Intuitively, the semantics for a class C is given by the set of those objects from the memory heap with type C. For a field $\mathsf{f} : \mathsf{C}'$ declared in a class C, the semantics of f are those pairs from the edge-set whose label is f.

Definition 2: Let $H = \langle V, E, L, r, p_1, \ldots, p_k, s_1, \ldots, s_n \rangle$ be a memory heap. Let $\mathsf{C}_1, \ldots, \mathsf{C}_j$ be the classes for objects in V. We define the semantics of class C_i $(1 \leq i \leq j)$ by $C_i = \{v \in V : v \text{ has type } \mathsf{C}_i\}$. Let class C_i $(1 \leq i \leq j)$ include a field $\mathsf{f} : \mathsf{C}'$. The semantics of field f in H, denoted by $[[\mathsf{f}]]_H$, is the function $f : C_i \rightarrow (C' \cup \{null\})$ defined by $[[\mathsf{f}]]_H = \{\langle o_1, o_2 \rangle \in E : L(\langle o_1, o_2 \rangle) = \mathsf{f}\}$.

A memory heap models a single test input. We will extend the notion of class field semantics to sets of memory heaps. Intuitively, we will join the semantics of all fields over all the heaps, into a single set of labeled pairs.

Definition 3: Given a test suite (set of memory heaps) $\mathcal{H} = \{H_1, \ldots, H_n\}$ and a field f, the semantics of f in $\mathcal{H}$, denoted by $[[\mathsf{f}]]_{\mathcal{H}}$, is defined as $\bigcup_{H \in \mathcal{H}} [[\mathsf{f}]]_H$. Let $\mathsf{F} = \{\mathsf{f}_1, \ldots \mathsf{f}_k\}$ be the set of fields in the class hierarchy for $\mathcal{H}$. We define the *class field semantics* of suite $\mathcal{H}$, denoted by $S_{\mathcal{H}}$, by $S_{\mathcal{H}} = \bigcup_{\mathsf{f} \in \mathsf{F}} [[\mathsf{f}]]_{\mathcal{H}}$.

Definition 4 below introduces field coverage criteria. Intuitively, any criterion based on populating the semantics of class fields (i.e., pairs $\langle o_1, o_2 \rangle$ labeled with the name of the field they are expected to belong to) in some specific way, is a field coverage criterion. A suite will satisfy the criterion if the semantics of class fields, as collected from the heap objects of the suite, cover field values as specified by the criterion.

Definition 4: 'Let $\mathsf{C}_1, \ldots, \mathsf{C}_n$ be a class hierarchy for a method m under test, i.e., method m may be executed in memory heaps whose objects are typed with types $\mathsf{C}_1, \ldots, \mathsf{C}_n$. Let S be a labeled set of object pairs, i.e., S is expected to be of the form $S = \{\langle l_1, r_1 \rangle, \ldots, \langle l_k, r_k \rangle\}$ satisfying, for each $1 \leq j \leq k$,

- $\langle l_j, r_j \rangle$ is labeled with a field f declared in $\mathsf{C}_1, \ldots, \mathsf{C}_n$,
- $\langle l_j, r_j \rangle$ is correctly typed as per the labeling, i.e., if pair $\langle l_j, r_j \rangle$ is labeled f, field f is declared in class C and has type C', then l_j has type C and r_j has type C'.

Field-coverage criterion FCC_S is satisfied by a test suite $\mathcal{T}$ iff the class field semantics (see Def. 3) $S_{\mathcal{T}}$ of heaps in $\mathcal{T}$ satisfies $S_{\mathcal{T}} \supseteq S$.

Figure 2 presents an example. Method $removeMin$ in class BinSearchTree removes the BinSearchTreeNode holding the least value in the binary search tree. Field coverage criteria are black-box, and therefore, as the example shows, we only need to know the interface of the method under test. The example presents a set S_0 that needs to be fully covered, i.e., the field-coverage criterion is FCC_{S_0}. The three memory heaps induced by the given trees (let us call these memory heaps H) cover set S_0 and therefore are a valid suite according to this testing criterion FCC_{S_0}. Another example, this time given by comprehension, is *non-null field coverage*; this criterion establishes that each field semantics must contain at least one pair whose second component is non-null. Any suite containing the second or third tree in Fig. 2 would satisfy this criterion, whereas a suite containing only the first one would not.

It is important to notice that memory heaps as we have described them require objects to be somehow identified since objects are the vertices of heap graphs. Selecting appropriate identifiers is important, and may be related to the programming language in which the software under test is provided. Languages in which the specific memory address where an object is stored is relevant (in particular, languages supporting pointer arithmetic) may lead one to choose object memory addresses as object identifiers. Notice that choosing such a concrete notion of object identifier can have a significant impact in how field-coverage criteria are described, as well as in how difficult it may be to guarantee that certain field-coverage criteria are

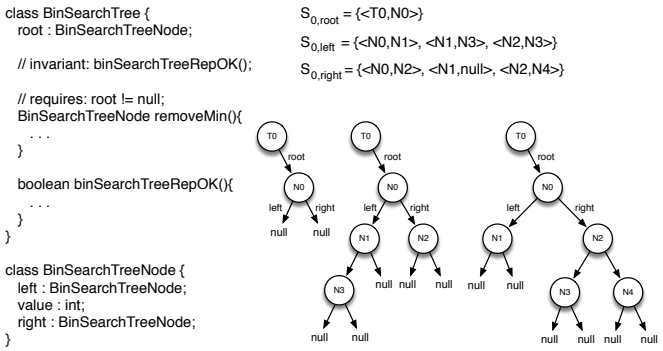

Fig. 2. Suite based on field coverage: an example.

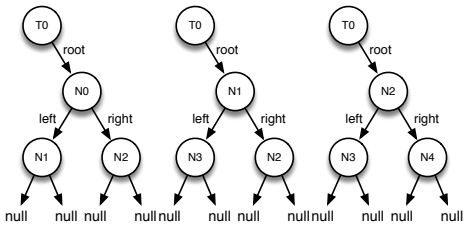

Fig. 3. Achieving field-base coverage by permuting object identifiers.

met. In this paper, we use a more abstract notion of object identifier, based simply on the position of each object in the breadth-first traversal of the heap. This is, in fact, the object identifier notion employed in Fig. 2 (where the order is given on a per-type basis). This choice not only simplifies in our case the definition of field coverage criteria; it also allows us to deal with *symmetry breaking*. Symmetry breaking requires canonical representations of memory heaps, thus guaranteeing that, if two heaps differ in their canonical representations, then they are non-isomorphic, or equivalently, if two memory heaps are isomorphic, their canonical representations will be the same. Symmetry breaking is important in test generation since it removes redundant cases, and mechanisms to guarantee it are available as part of various test generation tools. For example, Symbolic PathFinder [28] uses Lazy Initialization [15] to explore memory heaps, and in the process it removes symmetries. Similarly, Korat [5] generates non-isomorphic objects. TACO [10] uses automatically generated symmetry breaking axioms to remove symmetries. Notice that, if isomorphic structures are allowed, then one may achieve better coverage without actually considering different cases, e.g., the suite depicted in Fig. 3 satisfies FCC_{S_0} (see Fig. 2) by permuting objects identifiers in only one structure.

Another example of a field-coverage criterion is *field-exhaustive testing* [23]. In this criterion, the underlying set S of pairs to be covered is defined as containing all pairs that occur in *valid* (where validity is relative to some input's specification) memory heaps, bounded in size by a user-provided bound k. Field-exhaustive testing, as presented in [23], requires a *specification* of valid inputs, e.g., through a declarative precondition or a declarative object invariant. How "strong" such specification is, has an impact on the feasible pairs of field semantics (since these are restricted to those feasible in valid inputs).

Finally, let us remark that suites complying with field-coverage criteria do not need to be of minimal size. For instance, notice that the first tree in Fig. 2 may be safely removed, and the remaining two trees still conform a valid suite according to criterion FCC_{S_0}. Also, a field-coverage criterion may not lead to any valid suite, since it may require certain infeasible field semantics pairs to be covered. For instance, if we add to $S_{0,\mathsf{root}}$ the pair $\langle T_0, \mathsf{null}\rangle$, no memory heap satisfying the method precondition (`root != null`) may cover that pair.

III. Automated Generation of Field-Coverage Suites

Field-coverage, as defined in the previous section, is a very broad class of black-box coverage criteria. In this section, we will concentrate on how to automatically generate test inputs satisfying some specific field-coverage criteria. More precisely, we consider the use of field-coverage to reduce the number of symbolic paths during test generation via symbolic execution of a `repOK()` method, i.e., a routine that checks the representation invariant. In this case, the reduction is associated with imposing an estimated metric regarding field semantics, and prematurely stopping the systematic visit of symbolic paths when such metric is achieved. This approach then implies a *lossy technique*, in the terminology of [29], in the sense that it will compute an underapproximation of the set of required test inputs. Notice in particular that this work generalizes field-exhaustive testing, as presented in [23] since here we do not require formal declarative specifications to compute suites. Moreover, we will also present a series of test generation policies that do not demand *full* field-exhaustive suites but approximate these suites. In Section IV we will analyze the bug-finding capabilities of suites produced following these approaches, in comparison with the corresponding original techniques.

A. Approaching Field-Exhaustive Testing with TranScoping

The notion of *scope*, as a bounding parameter, is present in many automated analysis techniques. As a term, it originates in the context of Alloy [13], where it indicates the maximum number of atoms to be considered as part of signatures, thus making decidable the satisfiability of formulas in an undecidable logic (the scope makes domains finite, and thus formulas in relational logic can be "flattened" into propositional ones). This concept is directly mapped to other analysis tools, notably the test generation tools Korat [5] and TestEra [16], that borrowed the concept as well as the term. Other analysis techniques also require a scope of some kind:

- Models in model checking [8] need to be finite, and therefore bounds that limit the number of states are common.

- Test generation through symbolic execution, often following a depth-first traversal, imposes a maximum depth size for bounded path exploration.
- Software model checkers impose a maximum number of iterations and recursive calls.
- Random testing bounds the length of test sequences during test generation [21].

TranScoping [24] is a technique for estimating characteristics of yet to be run analyses for large scopes, based on data obtained from analyses performed in small scopes. Essentially, the idea is to perform *complete* analyses in small scopes, where they are still affordable, and extrapolate harvested data to larger scopes, that can be exploited to improve analysis, e.g., by identifying irrelevant parts of analysis that may be avoided, or selecting values for parameters of the analysis that showed better performance in small scopes. In this article, we will extrapolate information regarding field semantics, that will allow us to stop test input generation before it would naturally terminate, with the goal of producing test suites that approximate field-exhaustive ones and retain good coverage and bug finding capabilities.

Field-exhaustive suites cover all the pairs of object identifiers that may occur in valid (as per method preconditions, representation invariants, and scopes) memory heaps [23]. A problem that arises when computing test inputs through symbolic execution over an operational input specification, i.e., `repOK()`, is that *all* bounded symbolic paths must be traversed along the search of inputs that might contribute new pairs to the field semantics. Thus, even when our goal may be to achieve field-coverage, in practice one ends up generating a suite that guarantees bounded-paths coverage. Of course, some of the inputs generated along path traversal will not be added to the suite because they will be deemed superfluous (in case they do not contribute any new pairs to the field semantics), but still, the computational cost ends up being the same as that of path coverage.

As we will discuss in Section IV, when symbolically executing a `repOK()` invariant, it is possible to generate most of the required inputs early in the symbolic execution. Therefore, deciding a suitable condition to stop the execution in a way that good coverage/bug-finding is achieved, will allow us to reduce generation time considerably. We use tranScoping for this task.

1) TranScoping Field Size: Field-exhaustive suites cover those pairs of identifiers $\langle id_0, id_1 \rangle_{\mathsf{f}}$ such that $id_0.\mathsf{f} = id_1$ in some valid memory heap. A field-exhaustive suite then allows us to compute field f's semantics ($[[\mathsf{f}]]$), which is a set of pairs whose size we will note as $size_{\mathsf{f}}$. Therefore, given a class hierarchy for the method under test, from a field-exhaustive suite we will be able to extract field's semantics $[[\mathsf{f}_1]], \ldots, [[\mathsf{f}_k]]$, whose sizes will be noted $size_{\mathsf{f}_1}, \ldots, size_{\mathsf{f}_k}$, respectively. Notice that given a scope s for the number of objects, these sizes will vary depending on s. Therefore, for scopes $s_1, \ldots, s_n$ we can obtain families of sizes $size^{s_1}_{\mathsf{f}_1}, \ldots, size^{s_1}_{\mathsf{f}_k}, \ldots, size^{s_n}_{\mathsf{f}_1}, \ldots, size^{s_n}_{\mathsf{f}_k}$. In this setting, we will use the following tranScoping procedure. We will compute the actual sizes for scopes 1 through 5, i.e., compute the values $size^1_{\mathsf{f}_1}, \ldots, size^1_{\mathsf{f}_k}, \ldots, size^5_{\mathsf{f}_1}, \ldots, size^5_{\mathsf{f}_k}$. These values are computed by generating field-exhaustive suites for scopes 1 through 5; our rationale is that for these small scope values, field-exhaustive generation can be performed efficiently.

Algorithm 1 shows a first approach to suite computation, based on tranScoping size information. It receives as input an integer $smallScope$, which characterizes those scopes in the range 1 to $smallScope$ for which field-exhaustive suites will be computed. Also, it receives $Scope$, the scope for which a field-coverage suite will be generated. The algorithm assumes that class Analysis wraps a data generation tool such as Java/Symbolic PathFinder. We also assume this tool provides an iterator interface that allows us to ask for the "next" datum. Class FieldSizes stores, for each field f, the number of pairs that the inputs visited so far contribute to $[[\mathsf{f}]]$. Therefore, lines 3–6 store in array $fSizes$ the sizes of fields according to field-exhaustive suites for each of the scopes considered *small*, namely, scopes 1 to $smallScope$. Line 7, using an appropriate regression function (we will discuss such functions below), predicts the expected size of the fields' semantics in scope $Scope$. These predicted values will be used in line 11 to characterize the termination criterion of the while loop; the loop will iterate as long as any of the actual sizes so far stored in $fieldsSemantics$ has not reached the predicted value stored in $transcopedSizes$ (such is the meaning of operator $\triangleleft$). Inside the while loop (lines 12–19), a new input is retrieved from the analysis and the current semantics is updated with the aid of function updateSemantics. In case the new input actually contributes with new pairs to the field semantics, the input is stored as part of the generated suite. If there is no next input (lines 13–15), the algorithm terminates. The underlying testing criterion FCC_S that Algorithm 1 satisfies is that the size of $[[\mathsf{f}]]$ for each field is greater than or equal to the minimum between the actual field semantics size and the tranScoped field semantics size for the given *Scope*. We present a running example of Algorithm 1 further below.

We will consider three extrapolation methods in order to infer field sizes for larger scopes, namely,

- Using a linear regression line.
- Using a quadratic regression parabola.
- The function computed as the average of the functions above.

We will now discuss the reasons for choosing these regression functions. Linear functions usually underestimate the number of tuples in the field's semantics (cf. [23]). Yet on occasions, as may be the case for singly linked lists, they produce a good estimation. Also, even when they underestimate the number of tuples, this is useful when we aim at obtaining small test suites. Notice that the largest size a field's semantics may have, for scope n, is $n \times (n+1)$ (each one of the at most n objects in the field's domain may point to at most n objects or the null value). Therefore, a quadratic function may provide a good estimation of the actual field size. On occasions, though, the inferred value may be too close (or even exceed) the

Algorithm 1 Field-coverage test suite generation by field size tranScoping.

```
1: function COMPUTE-SUITE(int : smallScope, Scope)
2:     Analysis analysis = new Analysis();
3:     FieldSizes[] fSizes = new FieldSizes[smallScope];
4:     for (int i = 0, i < smallScope, i++) do
5:         fSizes[i] = computeFieldSizesForScope(i + 1, analysis);
6:     end for
7:     FieldSizes tranScopedSizes = tranScopeFieldSizes(fSizes, Scope);
8:     Set⟨Input⟩ suite = new Set⟨Input⟩;
9:     Map⟨FieldName, Pair ⟨Object, Object⟩⟩ fieldsSemantics =
10:        new Map⟨FieldName, Pair ⟨Object, Object⟩⟩();
11:    while tranScopedSizes ◁ sizesOf(fieldsSemantics) do
12:        Input input = getNextInput(analysis);
13:        if (input == NULL) then
14:            break;
15:        end if
16:        boolean addedPair = updateSemantics(fieldSemantics, input);
17:        if (addedPair) then
18:            suite.add(input);
19:        end if
20:    end while
21:    return suite;
22: end function
```

actual field size, leading to almost no profit in test generation time, compared to exploring the whole space of alternatives. Therefore, taking the average between the quadratic and the linear functions leads to a (still quadratic) function that grows slower than the original quadratic function. In Section IV we will evaluate input generation algorithms that use tranScoping to determine when generation must terminate, according to each of these stopping criteria.

2) TranScoping Generation Time: An alternative to tranScoping the size of the semantics of fields as a termination criterion is to consider the time required to reach the full-size field semantics. The algorithm for test suite generation is similar to Alg. 1, and is presented as Alg. 2.

Algorithm 2 Field-coverage test suite generation by time tranScoping.

```
1: function COMPUTE-SUITE(int : smallScope, Scope)
2:     Analysis analysis = new Analysis();
3:     Time[] times = new Times[smallScope];
4:     for (int i = 0, i < smallScope, i++) do
5:         times[i] = computeTimesForScope(i + 1, analysis);
6:     end for
7:     Time tranScopedTime = tranScopeTime(times, Scope);
8:     Set⟨Input⟩ suite = new Set⟨Input⟩;
9:     Map⟨FieldName, Pair ⟨Object, Object⟩⟩ fieldsSemantics =
10:        new Map⟨FieldName, Pair ⟨Object, Object⟩⟩();
11:    while elapsedTime < tranScopedTime do
12:        Input input = getNextInput(analysis);
13:        if (input == NULL) then
14:            break;
15:        end if
16:        boolean addedPair = updateSemantics(fieldSemantics, input);
17:        if (addedPair) then
18:            suite.add(input);
19:        end if
20:    end while
21:    return suite;
22: end function
```

Notice that unlike Alg. 1, which kept track of the size of the semantics of each field, in this case, we only need to keep track of the elapsed time of the current analysis in order to determine when generation must be stopped. The elapsed time is maintained by the variable $elapsedTime$. The underlying testing criterion FCC_S which Algorithm 2 satisfies is that each field contains the fraction of the actual field semantic discovered using the tranScoped time as a timeout. We present a running example of Algorithm 2 further below.

The extrapolation methods considered when tranScoping generation time will be the following:

- A quadratic regression parabola.
- An exponential regression.
- The function computed as the average of functions above.

We will now discuss the reasons for selecting these regression functions. The number of feasible paths in a program usually grows exponentially as the program size increases. Therefore, an exponential function may provide a good estimation of the actual time. However, the exponential function has the same limitations that the quadratic function had when tranScoping field size, i.e. inferring a value too close (or even exceeding) the actual generation time. On the other hand, a quadratic function may provide a suitable lower bound to flatten the average regression while providing a good underestimation of the actual time.

In order to compute the regression functions, we use the Ordinary Least Squares method (OLS) [11] to estimate the parameters of a multiple linear regression model. In particular, to compute an exponential regression we transform the observations to a linear model.

Table I presents a running example for BinSearchTree in order to illustrate how Algorithms 1 and 2 work. Each row corresponds to a different execution of the Symbolic PathFinder tool for the given scope. Columns **Suite size** and **Generation time** present the size of the generated suite and the time in seconds required for the generation, using 3 different test generation policies, namely, **Fet** (for field-exhaustive testing), **Size-a** (for Algorithm 1 using the average between the linear and quadratic regressions) and **Time-a** (for Algorithm 2 using the average between the quadratic and exponential regressions). Notice that the suite size and generation time for scopes 1 to 5 are the same for the three test generation policies since Algorithms 1 and 2 compute field-exhaustive suites for these scopes. Column $|[[\mathbf{f}]]|$ presents the actual field semantic sizes for fields *left* and *right* whereas $\mathbf{T}|[[\mathbf{f}]]|$ presents the tranScoped field semantic sizes using the average regression. Notice that in order to compute $|[[\mathbf{f}]]|$ we need to compute a field-exhaustive suite, while Algorithm 1 only needs to reach the bound $\mathbf{T}|[[\mathbf{f}]]|$, e.g., for scope 8 the generation will stop when the field semantics size of field *left* reaches 21 pairs *and* the field semantics size of field *right* reaches 25 pairs. This condition is met in 140 seconds and the generated suite contains 33 non-isomorphic structures whose bug-detection ability is comparable to that of field-exhaustive suites as we will see in section IV. Finally, columns $\mathbf{Time}$ and $\mathbf{T\ Time}$ present the time required to reach the full-size field semantics and the tranScoped time, respectively, using the average regression. Notice the difference between columns

TABLE I
RUNNING EXAMPLE WITH BINSEARCHTREE

	Suite size			Generation time			|[[f]]|	T|[[f]]|	Time	T Time
Scope	Fet	Size-a	Time-a	Fet	Size-a	Time-a	left/right			
1	2			1			1/1	-	1	-
2	4			2			3/3	-	2	-
3	7			2			5/6	-	2	-
4	11			3			8/9	-	2	-
5	16			8			11/13	-	4	-
6	22	20	17	31	9	9	15/17	14/16	9	5
7	29	25	17	136	32	9	19/22	17/20	32	6
8	37	33	17	641	140	9	24/27	21/25	142	8
9	46	38	22	2907	634	11	29/33	24/29	633	11

Generation time and **Time**. The former is the time required to generate a field-exhaustive suite, i.e. exploring the whole space of alternatives and the latter is the time required to reach the full-size field semantics, i.e. when it was the last time an input contributed new pairs to field semantics. Column **Time** can only be computed once the generation has finished. However, we can use these values from scopes 1 to 5 to make predictions for larger scopes which is the key concept of Algorithm 2, e.g., for scope 8 the generation will stop when the tranScoped timeout of 8 seconds is reached, generating a suite containing 17 non-isomorphic structures whose bug-detection ability is comparable to that of field-exhaustive suites as we will see in section IV.

3) Termination and Correctness of the Field-Coverage Suite Generation Algorithms: Termination of Alg. 2 follows immediately because the while loop execution time is bounded by $tranScopedTime$. Termination of Alg. 1, on the other hand, is not obvious. Notice that variable $tranScopedSizes$ may contain an inferred size for a field f that is larger than the actual maximum size for the semantics of said field. Therefore, the loop will only terminate in case getNextInput($analysis$) returns null. To this end, we will only consider analyses that, for a user-given scope, become finite-state. Of course, the number of states may be large, and in this case, the generation will terminate when the underlying analysis terminates. This is why the presented techniques, which allow us to predict when to stop the analyses, are essential. Regarding correctness, let us define set $S = \{\langle i, o\rangle_{\mathsf{f}} : \langle i, o\rangle \in fieldSemantics(\mathsf{f})\}$, i.e., for each field f, set S contains those pairs computed in variable $fieldSemantics$ with label f. It is then clear that variable $suite$ stores a test suite that satisfies testing criterion FCC_S.

IV. EXPERIMENTAL EVALUATION

In this section, we present the experimental details and we will evaluate several field-coverage criteria. Since our goal is to approximate field-exhaustive testing, we will compare, for each criterion, the achieved branch coverage and the mutation score, as well as the suite size and the generation time, against the values obtained for field-exhaustive testing. We will consider an implementation of the following subjects for our experimental evaluation:

(i) Sorted singly linked lists,
(ii) Binary search trees,
(iii) TreeSets,
(iv) AVL trees, and
(v) Binomial heaps.

A. Experimental details

Algorithm 3 shows the driver needed to generate inputs with Symbolic PathFinder (SPF) tranScoping field size. Line 3 instructs SPF to treat `X` as a symbolic variable, thus enabling lazy initialization over `X` itself as well as its fields. Line 4 prunes the search whenever an invalid instance is found (we generate suites of valid inputs). Symbolically executing this line would potentially spawn infinite paths since the size of a binary search tree is not restricted by the `repOK()`. However, this potentially infinite exploration is bounded by the scope of the analysis. The methods used in lines 5–8 are captured by the listener, except for `Verify.writeObjectToFile()` which is part of the Verify API of SPF. Method `addedPair()` checks whether the new input `X` contributes with new pairs to the field semantics. if so, line 6 solves the remaining path condition. Notice that the symbolic execution of `repOK()` only concretizes (as a consequence of lazy initialization) fields `left` and `right`. However, for basic types as the ones used in field `value` we must resort to the SMT-solver and request concrete values to generate the actual input. Line 7 serializes `X` and saves it to the unique path returned by `getNextPath()`. Finally, line 8 checks whether the termination condition holds and stops the analysis if so. Notice that tranScoping field size imposes a synchronic termination criterion, i.e finding an input could lead to prematurely stop the analysis, whereas tranScoping generation time imposes an asynchronous termination criterion in a timeout-wise manner.

In algorithms 1 and 2 we separated the computation of the suites in 2 stages for the sake of simplicity. The first stage where we call `tranScopeFieldSizes` and `tranScopeTime` for algorithms 1 and 2 respectively to estimate the target of each termination criterion and the second stage where we use these criteria to stop the analysis. This requires to run SPF twice, the first time up to scope `smallScope` and the second one up to the desired scope, however, there is no such separation in algorithm 3. As we discuss below, we imposed a search heuristic that explores inputs in ascending order, considering the size of the structures. That said, it is straightforward to take advantage of this property to compute the suite up to scope `smallScope`, tranScope the termination criteria and continue the analysis, running SPF only once.

To measure the coverage and mutation score we considered 3 different methods of each case study, namely `insert`, `remove` and `contains`. All of this methods take as input an integer which we selected in a bounded exhaustive manner, this is, to run the tests with a suite generated up to scope `n`, we considered the integers in the range [0, n-1]. The tests were written with the help of JUnit [31], and the coverage and mutation analyses were performed with Jacoco [32] and MuJava [19], respectively.

Algorithm 3 Driver to run SPF for binary search trees tranScoping field size.

```
1 public static void main(String[] args) {
2   BSTree X = new BSTree();
3   X = (BSTree) Debug.makeSymbolicRef("X", X);
4   Verify.ignoreIf(X == null || !X.repOK());
5   if (addedPair()){
6     solvePathCondition();
7     Verify.writeObjectToFile(X, getNextPath());
8     stopIfTerminationConditionHolds();
9   }
10 }
```

TABLE II
SUITE SIZE AND TIME TO GENERATE SUITES WITH SYMBOLIC PATHFINDER TRANSCOPING FIELD SIZE (s = SCOPE).

		size	time	size	time	size	time	size	time	size	time
		s = 6		s = 15		s = 25		s = 35		s = 45	
SortedList	fet	6	2	15	4	25	13	35	44	45	121
	fbt-l	6	2	15	4	25	13	35	43	45	120
	fbt-a	6	2	15	4	25	13	35	44	45	123
	fbt-q	6	2	15	4	25	13	35	43	45	121
		s = 6		s = 7		s = 8		s = 9		s = 10	
BSTree	fet	22	31	29	136	37	641	46	2907	56	TO
	fbt-l	18	9	23	31	27	33	31	138	37	142
	fbt-a	20	9	25	32	33	140	38	634	47	2955
	fbt-q	22	10	29	33	37	628	46	2950	56	TO
		s = 6		s = 7		s = 8		s = 9		s = 10	
TreeSet	fet	18	165	25	1025	33	6713	40	TO	-	-
	fbt-l	14	35	19	170	21	177	26	1048	-	-
	fbt-a	16	38	19	172	26	1036	31	1061	-	-
	fbt-q	18	42	23	181	29	1064	36	6812	-	-
		s = 8		s = 10		s = 12		s = 14		s = 15	
AvlTree	fet	28	44	41	199	58	817	85	5127	93	TO
	fbt-l	20	16	28	35	35	70	42	210	45	232
	fbt-a	24	23	35	70	47	371	62	849	70	1647
	fbt-q	28	35	41	202	58	813	85	4307	93	TO
		s = 6		s = 8		s = 10		s = 12		s = 13	
BinHeap	fet	7	17	9	109	11	633	13	3258	14	TO
	fbt-l	7	14	9	65	10	237	11	480	12	1461
	fbt-a	7	14	9	65	11	467	13	2235	14	6761
	fbt-q	7	17	9	107	11	627	13	3272	14	TO

The experiments were conducted on a 3.6GHz Intel Core i7 processor with 8 GB RAM, running archlinux 4.19. The replication package, as well as the complete results, can be found in https://sites.google.com/itba.edu.ar/fbt-ase19.

B. Results

In Tables II–VI we denote field-exhaustive testing by fet. We denote by fbt-l, fbt-q, and fbt-e the criteria that tranScopes a given attribute using a linear, quadratic or exponential regression function, respectively. The criterion that uses the average of the two corresponding regression functions is denoted by fbt-a.

In Tables II and III, times are expressed in seconds and a 2 hours timeout (marked as TO when reached) is set. In Tables V and VI, branch coverage and mutation score are expressed as percentages. Table IV shows the number of mutants generated for each case study. A total of 30 mutation operators were used for this task. The list of these operators can be found in the replication package.

TABLE III
SUITE SIZE AND TIME TO GENERATE SUITES WITH SYMBOLIC PATHFINDER TRANSCOPING GENERATION TIME (s = SCOPE).

		size	time	size	time	size	time	size	time	size	time
		s = 6		s = 15		s = 25		s = 35		s = 45	
SortedList	fet	6	2	15	4	25	13	35	44	45	121
	fbt-q	6	2	6	2	6	2	6	2	6	2
	fbt-a	6	2	15	4	6	3	35	44	45	123
	fbt-e	6	2	15	4	25	13	35	43	45	120
		s = 6		s = 7		s = 8		s = 9		s = 10	
BSTree	fet	22	31	29	136	37	641	46	2907	56	TO
	fbt-q	17	9	17	9	17	9	22	11	22	13
	fbt-a	17	9	17	9	17	9	22	11	22	12
	fbt-e	17	9	17	9	17	9	22	11	17	9
		s = 6		s = 7		s = 8		s = 9		s = 10	
TreeSet	fet	18	165	25	1025	33	6713	40	TO	-	-
	fbt-q	13	33	13	33	16	40	18	47	-	-
	fbt-a	13	33	13	34	13	33	18	43	-	-
	fbt-e	13	33	13	33	13	33	18	45	-	-
		s = 8		s = 10		s = 12		s = 14		s = 15	
AvlTree	fet	28	44	41	199	58	817	85	5127	93	TO
	fbt-q	14	9	17	11	18	14	20	19	22	21
	fbt-a	17	11	18	15	24	26	35	81	31	60
	fbt-e	17	11	22	20	28	38	35	72	35	98
		s = 6		s = 8		s = 10		s = 12		s = 13	
BinHeap	fet	7	17	9	109	11	633	13	3258	14	TO
	fbt-q	7	14	7	21	7	41	8	64	9	77
	fbt-a	7	14	7	18	7	31	9	85	8	64
	fbt-e	7	14	7	21	8	49	7	38	8	51

TABLE IV
NUMBER OF MUTANTS GENERATED FOR EACH CASE STUDY.

Case study	**# of mutants**
SortedList	595
BSTree	1280
TreeSet	7287
AvlTree	5145
BinHeap	10354

TABLE V
COMPARING COVERAGE AND MUTATION SCORE OF FIELD-EXHAUSTIVE AND FIELD COVERAGE SUITES TRANSCOPING FIELD SIZE (s = SCOPE).

		cov	mut	cov	mut	cov	mut	cov	mut	cov	mut
		s = 6		s = 15		s = 25		s = 35		s = 45	
SortedList	fet	27	85	27	86	27	86	27	86	27	86
	fbt-l	27	85	27	86	27	86	27	86	27	86
	fbt-a	27	85	27	86	27	86	27	86	27	86
	fbt-q	27	85	27	86	27	86	27	86	27	86
		s = 6		s = 7		s = 8		s = 9		s = 10	
BSTree	fet i	29	85	29	85	29	85	29	85	29	85
	fbt-l	29	85	29	85	29	85	29	85	29	85
	fbt-a	29	85	29	85	29	85	29	85	29	85
	fbt-q	29	85	29	85	29	85	29	85	29	85
		s = 6		s = 7		s = 8		s = 9		s = 10	
TreeSet	fet	33	72	33	72	33	73	33	73	-	-
	fbt-l	32	67	33	72	33	73	33	73	-	-
	fbt-a	33	72	33	72	33	73	33	73	-	-
	fbt-q	33	72	33	72	33	73	33	73	-	-
		s = 8		s = 10		s = 12		s = 14		s = 15	
AvlTree	fet	35	84	35	84	35	84	35	84	35	84
	fbt-l	35	84	35	84	35	84	35	84	35	84
	fbt-a	35	84	35	84	35	84	35	84	35	84
	fbt-q	35	84	35	84	35	84	35	84	35	84
		s = 6		s = 8		s = 10		s = 12		s = 13	
BinHeap	fet	24	74	25	75	25	75	25	82	25	84
	fbt-l	24	74	25	75	25	75	25	75	25	75
	fbt-a	24	74	25	75	25	75	25	82	25	84
	fbt-q	24	74	25	75	25	75	25	82	25	84

TABLE VI
COMPARING COVERAGE AND MUTATION SCORE OF FIELD-EXHAUSTIVE AND FIELD COVERAGE SUITES TRANSCOPING GENERATION TIME (s = SCOPE).

		cov	mut	cov	mut	cov	mut	cov	mut	cov	mut
		s = 6		s = 15		s = 25		s = 35		s = 45	
SortedList	fet	27	85	27	86	27	86	27	86	27	86
	fbt-q	27	85	27	86	27	86	27	86	27	86
	fbt-a	27	85	27	86	27	86	27	86	27	86
	fbt-e	27	85	27	86	27	86	27	86	27	86
		s = 6		s = 7		s = 8		s = 9		s = 10	
BSTree	fet	29	85	29	85	29	85	29	85	29	85
	fbt-q	29	85	29	85	29	85	29	85	29	85
	fbt-a	29	85	29	85	29	85	29	85	29	85
	fbt-e	29	85	29	85	29	85	29	85	29	85
		s = 6		s = 7		s = 8		s = 9		s = 10	
TreeSet	fet	33	72	33	72	33	73	33	73	-	-
	fbt-q	32	67	32	67	33	72	33	72	-	-
	fbt-a	32	67	32	67	32	67	33	72	-	-
	fbt-e	32	68	32	68	32	68	33	73	-	-
		s = 8		s = 10		s = 12		s = 14		s = 15	
AvlTree	fet	35	84	35	84	35	84	35	84	35	84
	fbt-q	35	82	35	83	35	83	35	84	35	84
	fbt-a	35	83	35	84	35	84	35	84	35	84
	fbt-e	35	83	35	84	35	84	35	84	35	84
		s = 6		s = 8		s = 10		s = 12		s = 13	
BinHeap	fet	24	74	25	75	25	75	25	82	25	84
	fbt-q	24	74	24	74	24	74	25	75	25	75
	fbt-a	24	74	24	74	24	74	25	75	25	75
	fbt-e	24	74	24	74	25	75	24	75	25	75

C. Discussion

Tables II and III show that the generation of test suites resorting to the presented techniques is most times significantly faster than generating field-exhaustive ones. Of course, such speedup would be worthless if the produced suites were of poorer quality. Interestingly, Table V shows that suites produced by tranScoping field size achieve, in almost all cases (all but 2) the same branch coverage and mutation score than field-exhaustive suites. As it may be expected (see Table VI), tranScoping suite generation time is less precise, although still achieves good branch coverage and mutation score. We believe the results obtained when tranScoping field size, are quite compelling.

There is an interesting observation to be made regarding sorted singly linked lists. Table II shows no gains for the proposed techniques. The reason for this phenomenon is twofold. On the one hand, as we said before, a linear function provides a good estimation when tranScoping field semantic sizes (indeed, the computed linear regression is exact) and the quadratic coefficient of the quadratic regression is almost 0 ($-3.31e-16$), which produces an almost linear function for the values of x in the range [1,50]. This causes the linear, the quadratic and the average regressions to produce the same (and exact) estimations. On the other hand, since there is only one valid instance per scope and this case study, in particular, is not fully benefited by the imposed search heuristic since it always requires a new node to produce a new instance and has no `previous` field (discussed below), we can not take advantage of the observation that most of the required inputs can be generated early in the symbolic execution.

Table III also shows an interesting result regarding singly linked lists. TranScoping generation time is, by definition, sensitive to execution times, and in this particular case, the generation up to scope `smallScope` usually takes less than 2 seconds. The lack of data as well as its variability, sometimes makes the OLS method to produce negative quadratic regression functions, stopping the analysis too early compared with the exponential regression.

Even though the last two observations are negative, Tables V and VI show these shortcomings have no effect over the quality of the generated suites, which in turn have the same coverage and mutation score as those generated using field exhaustive testing. The reason for this to happen is that the methods that handle singly linked lists are simple enough (they usually consider at most 3 different scenarios) to be covered and get their mutations killed even with a few inputs. The counterpart of singly linked lists, in terms of code complexity, are binomial heaps and the results obtained tranScoping time show that fewer inputs (Table III) negatively impact the mutation score, in particular for higher scopes (Table VI). However, the results obtained tranScoping size for the same case study are quite compelling.

As we said before, when symbolically executing a `repOK()` invariant, it is possible to generate most of the required inputs early in the symbolic execution. This observation has a significant impact on the presented techniques since it allows us to prematurely stop the search and generate a significative fraction of the inputs that would be generated using field-exhaustive testing. This is achieved in Symbolic Pathfinder by imposing a search heuristic that prioritizes the use of previously created objects and the null reference over new objects, as candidates when lazy initializing objects' fields. As a side effect, this heuristic explores inputs in order, considering the size of the structures.

The fact a `repOK()` procedural invariant is required may be perceived as a limitation of these techniques. However, as reported in [17], having such methods is a good programming practice. Alternatively, one may generate test inputs via symbolic execution of constructor methods. This alternative has 2 problems:

- It is not obvious how to combine these methods to explore all the possibilities (for instance, for red-black trees it is required to combine methods `insert` and `remove` in order to explore all valid tree nodes colorings).
- The symbolic execution of combinations of complex constructor methods, like `insert` and `remove` is by far more expensive than the symbolic execution of a `repOK()` method.

D. Threats to Validity

Experimental evaluations may be influenced by the selected subjects. To reduce bias, we selected the same subjects used in the evaluation of field-exhaustive testing in [23]. The selected subjects are good examples of programs with heavily constrained memory heaps with a broad range of complexity, going from those with simple invariants to some with complex constraints. Moreover, they are often used as benchmarks

in the evaluation of other analysis tools [5], [29]. A more thorough evaluation is pending, including more data structures, but also general purpose software using said data structures.

V. Related Work

Automated test generation is a research area that has received substantial attention in recent years, and important advances, leading to the development of many tools and techniques, have taken place. Some of the most effective and successful automated testing approaches are either based on random generation [6], [18], [21], evolutionary computation [9], model checking [29], constraint solving (including SMT [27] and SAT solving [1]) or some forms of exhaustive search [5].

Test suites generated by tools based on random generation and evolutionary computation such as Randoop [21], AutoTest [18], QuickCheck [6] and EvoSuite [9], generate *large* test suites. A consequence of such large suites is increased testing time, a problem we try to avoid with field-exhaustive testing. Our approach corresponds to *systematic* test generation, in the terminology of [26], which makes it closer to tools like Pex [27], FAJITA [1], Symbolic PathFinder [28] and Korat [5]. These tools are also *specification based*, i.e., they produce tests from input specifications, as opposed to tools like EvoSuite or Randoop, that use routines/methods of the tested subject to produce test sequences.

In [26] a set of experiments compare random testing and systematic testing for container classes, arriving at the conclusion that random testing produces suites that are comparable in quality (coverage, mutation score) to systematically produced suites (using shape abstraction) while consuming less computational resources (less generation time).

In our approach, computing field-exhaustive suites produces field extensions. These extensions are equivalent to (upper) *tight field bounds*, and thus our work is related to approaches to compute such bounds, e.g., [3], [10], [22]. However, none of these related approaches focuses on test generation; [10] does not collect instances produced along the tight bound computation process (and even if it did so, it would be significantly more inefficient than our approach, since it requires a cluster of computers for tight bound computation); [3], [22] follow a bounded exhaustive enumeration of instances to compute tight bounds, as opposed to our field-exhaustive mechanism.

VI. Conclusions

We proposed a generalization of the field-exhaustive testing criterion, that we called *field-coverage* testing criterion, whose satisfaction is associated with the coverage of feasible values for object fields. Besides formally defining the criterion, we developed an algorithm that automatically produces under-approximations of field-exhaustive suites using tranScoping, which estimates characteristics of yet to be run analyses for large scopes, based on data obtained from analyses performed in small scopes. The experimental results showed that tranScoping field size provides a suitable condition to prematurely stop the search while producing suites whose bug detection ability is comparable to those satisfying the field-exhaustive testing criterion, measured in terms of mutation score and branch coverage, in a fraction of the time.

Another conclusion we draw, is that field-exhaustive testing is a very powerful criterion which, even if weakened, still produces good coverage and mutation score.

References

[1] P. Abad, N. Aguirre, V. Bengolea, D. Ciolek, M. F. Frias, J. P. Galeotti, T. Maibaum, M. Moscato, N. Rosner and I. Vissani, "Improving test generation under rich contracts by tight bounds and incremental SAT solving", in Proceedings of Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, IEEE, 2013.

[2] P. Ammann and J. Offutt, "Introduction to software testing", Cambridge University Press, 2008.

[3] H. Bagheri and S. Malek, "Titanium: efficient analysis of evolving alloy specifications", Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016, 2016.

[4] K. Beck, "Embracing change with extreme programming", IEEE Computer 32(10), IEEE CS, 1999.

[5] C. Boyapati, S. Khurshid and D. Marinov, "Korat: Automated testing based on Java predicates", in Proceedings of International Symposium on Software Testing and Analysis ISSTA 2002, ACM, 2002.

[6] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs", in Proceedings of the fifth ACM SIGPLAN international conference on Functional programming ICFP 2000, ACM, 2000.

[7] M. Cohen, M. Dwyer and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy Approach", IEEE Transactions on Software Engineering. 34, 5, 633-650, 2008.

[8] E. Clarke, O. Grumberg and D. Peled, "Model Checking", MIT Press, 2000.

[9] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software", in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering ESEC/FSE 2011, ACM, 2011.

[10] J. P. Galeotti, N. Rosner, C. López Pombo and M. F. Frias, "Analysis of invariants for efficient bounded verification", in Proceedings of the 19th International Symposium on Software Testing and Analysis ISSTA 2010, ACM, 2010.

[11] F. Galton, "Regression Towards Mediocrity in Hereditary Stature", Journal of the Anthropological Institute, 15:246-263, 1886

[12] C. Ghezzi, M. Jazayeri and D. Mandiroli, "Fundamentals of Software Engineering", Second Edition, Prentice-Hall, 2003.

[13] D. Jackson, "Software Abstractions: Logic, language, and analysis", MIT Press, 2006.

[14] P. Jalote, "An Integrated Approach to Software Engineering", 3rd. Edition, Springer, 2005.

[15] S. Khurshid, C. Păsăreanu and W. Visser, "Generalized symbolic execution for model checking and testing", in Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2003, LNCS 2619, Springer, 2003.

[16] S. Khurshid and D. Marinov, "TestEra: Specification-Based Testing of Java Programs Using SAT", Autom. Soft. Eng. 11(4), Kluwer Academic, 2004.

[17] B. Liskov, "Program development in Java: abstraction, specification, and object-oriented design", Addison-Wesley, 2000.

[18] L. Liu, B. Meyer and B. Schoeller, "Using contracts and boolean queries to improve the quality of automatic test generation", in Proceedings of the 1st International Conference on Tests and Proofs TAP 2007, LNCS 4454, Springer, 2007.

[19] Y.-S. Ma, J. Offutt and Y.-R. Kwon: "MuJava: An Automated Class Mutation System", Journal of Software Testing, Verification and Reliability, 15(2), 2005

[20] S. Nelson, "Certification processes for safety-critical and mission-critical aerospace software", Report NASA/CR-2003-212806, Ames Research Center, 2003.

[21] C. Pacheco, S. K. Lahiri, M. D. Ernst and T. Ball, “Feedback-directed random test generation”, in Proceedings of the 29th international conference on Software Engineering ICSE 2007, IEEE, 2007.

[22] P. Ponzio, N. Rosner, N. Aguirre and M. Frias: “Efficient tight field bounds computation based on shape predicates”, Lecture Notes in Computer Science FM 2014: Formal Methods. 531546, 2014.

[23] P. Ponzio, N. Aguirre, M. Frias and W. Visser, “Field-Exhaustive Testing”, in Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering FSE 2016, ACM, 2016.

[24] N. Rosner, C. G. López Pombo, N. Aguirre, A. Jaoua, A. Mili and M. Frias, “Parallel Bounded Verification of Alloy Models by TranScoping”, in Proceedings of the 5th International Conference on Verified Software: Theories, Tools, Experiments VSTTE 2013, LNCS 8164, Springer, 2013.

[25] D. Shao, S. Khurshid and D. E. Perry, “Whispec: White-box testing of libraries using declarative specifications”, Proceedings of the 2007 Symposium on Library-Centric Software Design - LCSD ’07, 2007.

[26] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser and D. Marinov, “Testing container classes: random or systematic?” Fundamental Approaches to Software Engineering Lecture Notes in Computer Science. 262-277, 2011.

[27] N. Tillmann, J. de Halleux, “Pex: White Box Test Generation for .NET”, in Proceedings of the 2nd International Conference on Tests and Proofs TAP 2008, LNCS 4966, Springer, 2008.

[28] W. Visser, C. S. Pasareanu and S. Khurshid, “Test Input Generation with Java PathFinder”, in Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis ISSTA 2004, ACM, 2004.

[29] W. Visser, C. S. Pasareanu and R. Pelánek, “Test Input Generation for Java Containers using State Matching”, in Proceedings of the International Symposium on Software Testing and Analysis ISSTA 2006, ACM, 2006.

[30] H. Zhu, P. Hall and J. May, “Software Unit Test Coverage and Adequacy”, ACM Computing Surveys 29(4), ACM, 1997.

[31] (2019, Sep.) JUnit. [Online]. Available: https://junit.org/

[32] (2019, Sep.) Jacoco. [Online]. Available: https://www.eclemma.org/jacoco/